Appears in 10th ACM/IFIP/USENIX International Middleware Conference, Industrial Track, Urbana-Champaign, IL (December 2009)

Drivolution: Rethinking the Database Driver Lifecycle

Emmanuel Cecchet
University of Massachusetts and Aster Data Systems, Inc.
cecchet@cs.umass.edu

George Candea
EPFL and Aster Data Systems, Inc.
george.candea@epfl.ch

ABSTRACT

The current design of database drivers – a necessary evil for interacting with a DBMS – imposes undue burdens on those who install, upgrade, and manage database systems and their applications. In this paper, we introduce Drivolution, a new architecture for DB drivers that reduces the cost, risk, and downtime associated with driver distribution, deployment and upgrade in large production environments.

We view DB drivers as an integral part of the DB schema, so Drivolution stores drivers in the database itself. Drivers are dynamically downloaded and installed by a small bootloader that resides within each client applications. Downloading, installing, and upgrading drivers occurs transparently to applications, and existing DB management mechanisms are used to define and enforce desired security policies. We show how Drivolution can be integrated into legacy DB engines, replication middleware, and applications, without requiring changes to the server or client applications. We present several case studies that illustrate the use of Drivolution in production environments.

1. INTRODUCTION

Despite the standardization of database APIs, the heterogeneity of servers and application platforms is daunting. For example, the MySQL DBMS [7] officially supports Connector/NET, Connector/ODBC, Connector/J (Java), Connector/MXJ, Connector/PHP, mysqlclient (C API), mysqli (PHP), DBD::mysql (Perl), MySQLdb (Python), DBD::MySQL & ruby-mysql (Ruby) and MySQL++ (C++). This does not include independently developed APIs such as TCL or Eiffel wrappers.

Such heterogeneity poses a significant challenge in large production environments that evolve over time. It is common to see a large number of diverse client applications, even if they all access a single database instance. Merely upgrading the client side drivers for the one database can turn into a complex problem spanning multiple architectures and platforms and requiring a broad set of skills and expertise among the operations staff.

Even starting out with a single database version on a single platform, large deployments inevitably become heterogeneous over time, as they evolve to meet business needs. The diversity of drivers is thus compounded by the heterogeneity of database servers and platforms on which they run; for example, MySQL is officially supported on 63 different platforms [8].

The problem becomes even more acute in replicated DB environments, where upgrading database drivers on DBMS clients easily becomes a more complex problem than upgrading the database itself, because it needs to take into account the Cartesian product of the set of drivers and the set of databases running in the organization. This complexity is a challenge in hosting centers and large web sites. For example, Pair Networks' 500 web servers host many applications (in PHP, Ruby, Perl, etc.) that access 100 MySQL databases [11], and Match.com has more than 100 web servers accessing a single database cluster of only a few machines [6].

We see four major problems that have an important practical impact on large production environments: (1) driver distribution is separate from the database engine, which can easily lead to incompatibilities and mismatches between drivers and databases, (2) driver installation requires manual operations on each client machine, (3) driver upgrades are disruptive and require applications to be reconfigured and restarted, and (4) malicious applications can use specifically crafted drivers to exploit security holes in the database specific network protocol or attack database servers with buffer overflow techniques. These combined issues lead to high operational costs and foregone revenue due to application downtime.

In this paper we describe Drivolution, an alternative to the traditional database driver architecture. With Drivolution, drivers are stored in the database or the replication middleware and are distributed by the server to its clients on-demand. This way, driver distribution and deployment is managed from a centralized location, closely associated with the database. Clients are guaranteed to get the correct driver version to access the desired database. A generic client-side bootloader downloads and executes the driver code provided by the database. This bootloader is simple and almost never needs upgrading, much like an operating system bootloader. The bootloader can download multiple drivers and switch a client from one version to another to achieve driver upgrades that are transparent to the application.

We implemented Drivolution for the popular JDBC API and experimented with it in the context of the Sequoia [12] database clustering middleware. We show how Drivolution can be used in various configurations to provide seamless driver distribution, installation and upgrades for large scale setups involving legacy applications and databases. The ability of Drivolution to co-exist with current technologies and legacy components offers a smooth path for making Drivolution a core service of DBMSes. We hope to motivate adoption of the Drivolution architecture in other production DBMSes. Drivolution is freely available [3].

Middleware'09, Nov. -Dec. 2009, Urbana Champaign, Illinois, USA.

The rest of this paper is structured as follows: Section 2 describes the current state-of-the-art in database drivers and the issues that arise from their current lifecycle. Section 3 and 4 present the Drivolution concepts and design. Section 5 gives multiple use cases of Drivolution with the Sequoia database replication middleware and discusses the pros and cons of each configuration for driver installation and upgrades in production environments. Section 6 concludes the paper.

2. CURRENT STATE-OF-THE-ART

State-of-the-art databases today have adopted a fairly uniform way of managing DB drivers. The typical lifecycle is:

1. *Get an appropriate driver package from vendor*
2. *Install the driver on the client application machine*
3. *Configure the client application to use the driver*
4. *Start the application and load the database driver*
5. *Connect to database and check protocol compatibility*
6. *Authenticate*
7. *Execute requests*

A driver update requires the following steps:

8. *Stop the application*
9. *Uninstall old driver*
10. *Repeat steps 1 through 7*

There are considerably more driver instances deployed than DBMS instances. First, there are many more client applications accessing databases than database instances executing queries. Second, as applications can run on a large variety of platforms and middleware, drivers generally have to support many more architectures than database engines, which are designed for the major operating systems and hardware architectures.

Driver diversity is a major challenge in practice. Drivers provide support for different programming languages and APIs. Diversity also arises from the lack of standardization in communication protocols between client application and database engines. Despite several efforts [9], many protocols still co-exist in most enterprise setups.

Step 1 above requires the application developers to know in advance which database version is going to be used at deployment, if they want to ship the application with the driver. This might not be possible a priori or the driver licensing terms might not allow its redistribution with an application. Any resulting version mismatch between the driver and the database would prevent the application from accessing its database.

Steps 2 and 3 can be relatively easy if the machine hosting the application connects to a single database and the driver comes in a package that automates the installation process. If an application needs to access multiple databases using different driver versions, installation and configuration quickly become complex. Each driver implementation needs to be loaded in a separate namespace and this can be an issue if the drivers have not been designed to co-exist in heterogeneous environments. Driver settings incompatible with database settings also prevent proper interactions with the database.

It is only during step 4 that the compatibility between the application and the driver is tested. The main sources of incompatibility are mismatches between the binary format of the driver and the hardware platform or incompatible compilation/linking options between the driver and application.

Step 5 is where the compatibility between the database and the driver is checked. Note that not all implementations check their interoperability at the protocol level. In such cases, incompatibility errors might be detected at an even later stage than connection time.

Step 6 can introduce additional errors if the driver does not support authentication methods that are required by the database. It is only if this step is successful that the application can finally issue requests to the database.

Driver updates are not necessarily released simultaneously for all platforms. This means that large setups with many applications accessing a database have to perform updates one by one for each application. The process requires the application to be stopped for the driver to be updated. If the upgrade is not automatic or does not support the specifics of the installation (multiple versions installed, multi-database setups, etc…) a manual uninstall must be performed first before re-iterating through all the steps (1 to 7). Not only is the update process disruptive for the application, but it is also error prone, which lengthens the downtime of the application. Often, software upgrades are delayed or simply not applied because of their complexity or the risks associated with the process. This can leave potential security holes open and compromise both applications and databases.

3. DRIVOLUTION DESIGN

We propose a new design for managing the lifecycle of database drivers that simplifies their installation, deployment and management on client machines. Figure 1 gives an overview of the Drivolution architecture.

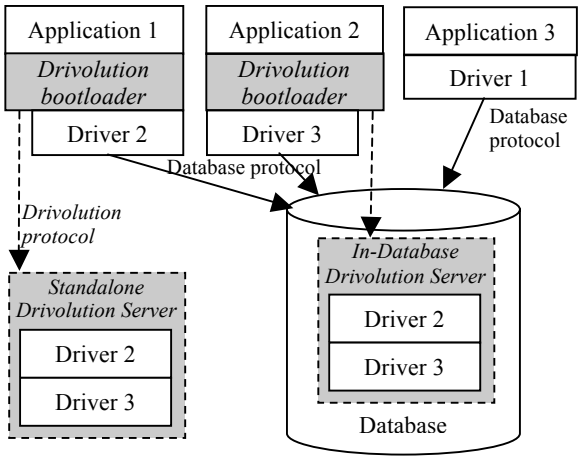

Figure 1. Drivolution architecture overview

3.1 Overview

In Drivolution, drivers are normally stored in the database in a regular table. Alternatively, a standalone external Drivolution server can be used as a service to distribute drivers. Small, stripped-down Drivolution bootloaders are used by client applications to interface to a Drivolution Server module to download the appropriate driver code corresponding to the database. Unlike drivers, Drivolution bootloaders hardly ever need to be updated due to their simple and limited feature set (Section 3.1.1). Drivolution servers implement the Drivolution protocol (Section 3.4) and are separate from the database

protocol. This allows applications that do not use Drivolution to still access the database with a conventional driver like Application 3 in Figure 1.

Drivolution uses leases to limit the time for which distributed drivers are valid, like DHCP does for IP addresses. The Drivolution bootstrap protocol is inspired from DHCP and has only three messages: DRIVOLUTION_REQUEST, DRIVOLUTION_OFFER and DRIVOLUTION_ERROR. The driver file transfer can use an FTP-like protocol or an encrypted and authenticated equivalent in insecure environments.

The Drivolution bootloader must first send a DRIVOLUTION_REQUEST message (to a specific server or to a list of trusted servers) containing the name of the database and corresponding credentials, the API name (e.g. JDBC, ODBC…) with an optional version, the client platform (e.g. JRE 1.5, windows-i586, linux-x86_64…) and additional options in case multiple drivers matching the previous criteria are available.

Based on the information received, the Drivolution Server queries its information schema to find the appropriate driver. Further information on driver match making can be found in the Drivolution documentation [1]. If no driver can be found, a DRIVOLUTION_ERROR message is sent back with an optional detailed error message in plain text (invalid database, no driver for specified API/platform, etc…). If multiple drivers match the request, the first matching driver is chosen. A DRIVOLUTION_OFFER message is then sent back to the bootloader. The message contains the lease time, the driver location and format. The driver is then downloaded.

The transfer can be secured using an SSL channel, in which case the bootloader verifies the Drivolution server's SSL certificate (to make sure the server is legitimate) and the driver cannot be tampered with during transfer by a potentially malicious middleman. It is also possible to sign drivers, and have a separate trusted wrapper in the bootloader verify signatures.

In the case of replicated databases, multiple replicas can answer client requests and this is also true for Drivolution. As for DHCP, a DRIVOLUTION_DISCOVER message can be broadcast to the network with the same information as a request message. All Drivolution servers that have an appropriate driver send a DRIVOLUTION_OFFER message back. The bootloader can then send a unicast DRIVOLUTION_REQUEST to one of the Drivolution servers. This mechanism allows databases to be added or removed from a database cluster in a decoupled manner, without having to reconfigure client applications.

The DHCP-like protocol employed by Drivolution offers a tradeoff between manageability and security: on one hand, it makes the distribution of drivers easy; on the other hand, it exposes Drivolution to man-in-the-middle attacks. At the same time, unencrypted driver transfer channels offer opportunities for drivers to be replaced with malicious ones. In its default configuration, Drivolution uses encrypted authenticated SSL channels (described above). It is important for administrators to understand that switching to a less secure configuration can lead to serious system compromise, and this risk is often not worth it.

3.1.1 Drivolution Bootloader

The Drivolution bootloader is an interceptor that substitutes the driver in the client application. It simply intercepts the connect method call of the API to capture the necessary information to retrieve the driver from the Drivolution Server in the DBMS. Once the driver code has been transferred, it is loaded dynamically into the application's memory. The application can then transparently use the driver, without consideration of how installation occurred.

Dynamic code loading may not be available in all languages on all platforms, but it can be implemented securely in most popular environments such as Java [4], .NET [13], C++ [10] or Perl using DynaLoader [2]. Connection configuration options are passed to the installed driver, which allows the application to continue to use database driver specific options or extensions. All other calls are passed through to the driver. Connection options can also be configured and enforced on the Drivolution server, which then sends a pre-configured driver to the client.

Drivolution bootloaders are generic to the extent that only one implementation per API and platform is needed. They are database or driver implementation neutral. For example, we have implemented a single Drivolution JDBC bootloader in Java [3] that supports all JDBC drivers of all databases on all platforms. It has the ability to load multiple implementations of drivers and to switch from one implementation to another, so that new connect calls can use a more recent driver version.

Bootloaders are only designed to support one fixed API and do not support migration among different APIs. However, as only the connection establishment part of the API is intercepted, bootloaders do not have to be changed if other parts of the API are changed. API changes in Drivolution are not a limitation for high availability, as the application would have to be changed to use the new API anyway. Drivolution is meant for applications that need to dynamically upgrade their drivers while keeping the same database API.

Dynamic driver updates that are transparent to applications may tempt administrators to deploy updates without rigorous testing. However, good testing practices are at the same time easier to implement. For example, a new driver version could be deployed to a single client machine with a short lease; if it works correctly, then it can be deployed more widely.

3.1.2 Updating the Driver

If the application in which the bootloader is hosted has not terminated before the lease has expired, the bootloader contacts the Drivolution Server to either renew its lease or get a new version of the driver by resending a DRIVOLUTION_REQUEST message. This allows Drivolution bootloaders to poll regularly for driver updates in critical applications that are never stopped.

When the driver needs to be upgraded, three replacement policies are available to transition existing connections (more details in Section 3.3). New connections always use the most recent downloaded driver. Existing connections using the old driver must be terminated before transitioning to the new driver. Depending on the policy, existing connections remain active until they have terminated their current transaction or until they are explicitly closed by the application, or forced to close.

When the lease has expired, but no new driver is available for replacement, a DRIVOLUTION_ERROR is sent back. The policy to close active connections is based on the current lease. Existing connections can remain active with the revoked driver until they terminate by an explicit closing by the application. In that case, the bootloader blocks new connection requests and it returns errors explaining the absence of a suitable driver.

3.2 Driver Lifecycle in Drivolution

Drivolution offers a simpler lifecycle than the current state-of-the-art described in Section 2. It consists of the following steps:

1. *Get an appropriate Drivolution bootloader*
2. *Install the Drivolution bootloader on the client application machine*
3. *Configure client application to use Drivolution bootloader*
4. *Start the application*

When a driver update is needed, all clients can be upgraded in a single step:

1. *Add new driver to the Drivolution Server*

The number of steps required for installation is reduced since, once the bootloader has been installed, all incompatibilities between database driver and server are avoided. The upgrade process drops from ten steps per client application to one simple insert operation on the Drivolution Server.

Driver upgrades are provided typically by database vendors. The database administrator (DBA) is responsible for database and driver upgrades. The Drivolution server can provide additional sanity checks to help the DBA make sure that newly installed drivers are compatible with the current database. For example, the upgrade can be performed on a test machine and then pushed to all other machines.

Effective driver renewal on the client side depends on the lease time that has been chosen. The first lease can be set to be very short and, if there are problems, the administrator can revert the driver in the Drivolution server. Shorter lease times allow faster reaction to upgrades but higher traffic to the Drivolution Server. Settings ranging from an hour to a day are suitable. Alternatively, a dedicated channel between the Drivolution bootloader and Server allows the Drivolution Server to immediately signal that a new driver is available. Revoking connections can be performed by the bootloader or enforced in the database server, if the Drivolution Server is tightly integrated with the database engine.

A misconfiguration or unavailability of the Drivolution Server can impact a large number of applications, similar (in the worst case) to a database outage. Note that the Drivolution Server can be replicated and a failure should have a minimal impact on already running applications since it only impacts new driver requests or driver renewal requests.

3.3 Schema for In-Database Drivers

We view drivers as being part of the database schema, and thus they belong to the database system tables.

We extend existing database information schema with a table that stores drivers and their metadata. This way, no new development is required and standard database mechanisms can be used to store drivers in the database. New drivers can be installed using simple INSERT statements and retrieved using regular SELECT queries. Table 1 describes a definition of the driver table that can be stored in the database information schema. Data type definitions follow the ANSI SQL 2003 standard. Each driver supports a specific set of APIs and platforms such as JDBC3 on JRE 1.5 or ODBC 3.5 on linux_x86_64. NULL values for API version numbers or platform specifications mean that all versions or platforms are supported, respectively. The driver version number is optional.

Table 1. Information schema driver table definition

Column name	Data type	Description
driver_id	INTEGER NOT NULL PRIMARY KEY	Primary key identifying drivers (to be used as a foreign key by other information schema tables for integrity checks)
api_name	VARCHAR NOT NULL	Supported API name (e.g. JDBC, ODBC…)
api_version_major	INTEGER	API major version number
api_version_minor	INTEGER	API minor version number
platform	VARCHAR	Name of the platform(s) supported
driver_version_major	INTEGER	Driver major version number
driver_version_minor	INTEGER	Driver minor version number
driver_version_micro	INTEGER	Driver micro version number
binary_code	BLOB NOT NULL	Binary of the driver code
binary_format	VARCHAR NOT NULL	Format of the binary code (e.g. JAR, ZIP…)

Standard database security mechanisms can be used to limit access to this table to a specific set of users or client IP addresses. Furthermore, to refine the management operations, we add a *driver_permission* table to the information schema that defines access rights and update policies for drivers. It would be possible to expand the database GRANT command to handle such policies.

Table 2. Driver_permission table description

Column name	Data type	Description
user	VARCHAR	User name
client_ip	VARCHAR	IP address of the client
database	VARCHAR	Database name
driver_id	INTEGER NOT NULL REFERENCES driver(driver_id)	Identifier of the driver in the driver table
driver_options	VARCHAR	Driver configuration options
start_date	TIMESTAMP	Date from which the driver can be downloaded
end_date	TIMESTAMP	Date until which the driver can be downloaded
lease_time_in_ms	BIGINT	Maximum lease time in ms
renew_policy	INTEGER 0: RENEW 1: UPGRADE 2: REVOKE	Policy to apply when a lease needs to be renewed.
expiration_policy	INTEGER 0: AFTER_CLOSE 1: AFTER_COMMIT 2: IMMEDIATE	Policy to apply when lease has expired (encoded as an integer).
transfer_method	INTEGER -1: ANY >=0: Protocol id	Transfer protocol to use to download the driver code.

Table 2 presents the *driver_permission* table. It defines which client gets which driver for each database instance. This is especially useful when different database instances require

different extensions, e.g. GIS (Geographic Information System), NLS (National Language Support), or specific authentication methods, and thus different drivers.

Additional client specific configuration options (*driver_options*) can be given to instruct the bootloader to enforce particular settings at driver loading time. The validity of a driver can be defined by dates (i.e. *start_date* and *end_date*) or by a lease time after which the bootloader has to recheck if a new version of the driver is available. The method used to transfer the driver code can be restricted to a specific secure protocol or use any protocols supported by the bootloader and the Server. The protocols are details in the Drivolution documentation [1].

When the lease has expired and must be renewed, the *renew_policy* defines the action the bootloader must take. It can continue to use the same driver (RENEW), download a new driver (UPGRADE) or terminate to use the current driver even though there is no replacement available (REVOKE). The *expiration_policy* parameter defines when the renew policy must be applied. The options are to wait for all current connections to be closed (AFTER_CLOSE), terminate connections as soon as they have committed their in-flight transactions (AFTER_COMMIT) or terminate immediately (IMMEDIATE).

3.4 Drivolution Protocol

The Drivolution protocol is used by the Drivolution bootloader to negotiate the appropriate DB driver with the database. A complete specification of the protocol and an open source Java implementation can be found on the Drivolution web site [3].

3.4.1 Getting the Appropriate Driver

Table 3 describes the Drivolution bootstrap protocol in general terms. For clarity, we omit the details regarding encryption and signature verification.

Table 3. Drivolution bootstrap protocol description

Drivolution bootloader	Drivolution Server
send(host, port, DRIVOLUTION_REQUEST)	

	if no driver matching request {
	send(DRIVOLUTION_ERROR)
	} else {
	send(DRIVOLUTION_OFFER)
	}

FILE_REQUEST(driver_file)	
	FILE_DATA(binary_code)
recheck_time = current_time + expiration_time_in_ms decode(binary_format,binary_code) load(decoded_binary_code)	

The Drivolution bootloader must first open a connection to the DBMS and then send a DRIVOLUTION_REQUEST message. The message contains the following information:

- *name of the database to be accessed with optional user/password information if authentication is required,*
- *API name (e.g. JDBC, ODBC…) with an optional version,*
- *client_platform (e.g. JRE 1.5, windows-i586, linux-x86_64…) on which the bootloader is running,*
- *optional preferred binary format and driver version number in case multiple drivers matching the previous criteria are available.*

Based on the information received, the Drivolution Server queries the information schema to find the appropriate driver (see section 4.1.1). If no driver can be found, a DRIVOLUTION_ERROR message is sent back with an optional detailed error message in plain text (invalid database, no driver for specified API/platform, etc…). If multiple drivers match the request, the first matching driver is chosen. A DRIVOLUTION_OFFER message is then sent back to the bootloader. The message contains one of the three expiration policies presented in section 3.3 along with the lease time, the driver location and format. The driver is then downloaded using a transport protocol that can be secured corresponding to the operating environment.

Table 4. Drivolution lease renewal protocol description

Drivolution bootloader	Drivolution Server
if (current_time >= recheck_time) send(host, port, DRIVOLUTION_REQUEST)	

	if (**driver still valid**) {
	send(DRIVOLUTION_OFFER)
	} else if (**new driver available**){
	send(DRIVOLUTION_OFFER)
	FILE_DATA(binary_code)
	} else { // **no driver available**
	send (DRIVOLUTION_ERROR)
	}

if (renew_policy == **RENEW**) {
recheck_time = current_time + expiration_time_in_ms
} else if (renew_policy == **UPGRADE**) {
FILE_REQUEST(driver_file)
recheck_time = current_time + expiration_time_in_ms
decode(binary_format, binary_code)
load(decoded_binary_code)
connect_use_new_driver
switch (expiration_policy) {
case **AFTER_CLOSE**:
wait_for_active_connections_closing
break;
case **AFTER_COMMIT**:
close_active_connections_after_commit
break;
case **IMMEDIATE**:
terminate_all_active_connections
break;
}
unload_old_driver
} else if ((renew_policy == **REVOKE**) ||
DRIVOLUTION_ERROR) {
switch (current_expiration_policy)
case **AFTER_CLOSE**:
disable_new_connections
wait_for_active_connections_closing
break;
case **AFTER_COMMIT**:
disable_new_connections
close_active_connections_idle_or_after_commit
break;
case **IMMEDIATE**:
terminate_all_active_connections
break;
}
unload_old_driver
}	

3.4.2 Driver update

If the application in which the bootloader is hosted has not terminated before the driver validity has expired, the bootloader contacts the Drivolution Server to either renew its lease or get a new version of the driver by resending a DRIVOLUTION_REQUEST message. This allows Drivolution bootloaders to poll regularly for driver updates in critical applications that are never stopped. bootloaders can use a dedicated thread as a timer to contact the Drivolution Server as soon as the timer expires, or they can wait lazily for an application call to trigger the check.

Table 4 describes the driver renewal protocol. If the driver can be kept for a new lease, a DRIVOLUTION_OFFER without data file instructs the bootloader to continue to use the same driver. When the driver needs to be upgraded, three replacement policies are available to transition existing connections. New connections always use the most recent downloaded driver. Existing connections using the old driver must be terminated before transitioning to the new driver. Depending on the policy, existing connections remain active until they are explicitly closed by the application (AFTER_CLOSE), or closed as soon as they are idle or have terminated their current transaction (AFTER_COMMIT), or are forced to close immediately (IMMEDIATE). If the client uses a connection pool, the first option might not be a good choice since connection renewal is highly dependent on connection pool settings and application load.

When the driver has expired but no new driver is available for replacement, a DRIVOLUTION_ERROR is sent back. The policy to close active connections is based on the current lease. Existing connections can remain active with the revoked driver until they terminate by an explicit closing by the application (AFTER_CLOSE policy). In that case, the bootloader blocks new connection requests and it returns errors explaining the absence of a suitable driver. The other policies terminate immediately all client connections (IMMEDIATE) or as soon as they are idle or their current transaction completes (AFTER_COMMIT).

4. DRIVOLUTION FOR LEGACY DATABASE SERVERS

The Drivolution Server can clearly be implemented "from scratch" as a new service of an DBMS engine, but it is also fairly easy to provide Drivolution for legacy DBMSes. After an overview of the server-side logic (Section 4.1.1), we present the design of the in-database Drivolution server (Section 4.1.2) and database-external Drivolution server (Section 4.1.3). We also describe how Drivolution can run as a standalone service for multiple DBs (Section 4.1.4).

4.1.1 Server Logic

The Drivolution Server side logic is relatively simple. Sample code 1 shows the SQL statement to retrieve the appropriate driver based on client preferences.

```
SELECT binary_format, binary_code
FROM information_schema.drivers
WHERE api_name LIKE $client_api_name
AND (platform IS NULL
     OR platform LIKE $client_platform)
AND ($client_api_version IS NULL
     OR api_version IS NULL
     OR $client_api_version LIKE
        api_version)
AND ($client_driver_version IS NULL
     OR driver_version IS NULL
     OR $client_driver_version LIKE
        driver_version)
```

Sample code 1. SQL request to retrieve driver based on client preferences

If this statement is unsuccessful, a simple SELECT without preferences (omitting the part of the statement in italics) can be issued. If this statement does not return any row, then it means that no driver is available for that client.

If the server contains a distribution table as described in Section 3.3, then that table should be queried first using the statement illustrated in Sample code 2. This gives a short list of available drivers for that client. This list can be further sorted with client preferences as explained above.

```
SELECT driver_id
FROM information_schema.distribution
WHERE (database IS NULL
       OR database LIKE $user_database)
AND (user IS NULL
     OR user LIKE $client_user)
AND (client_ip IS NULL
     OR client_ip LIKE $client_client_ip)
AND (start_date IS NULL
     OR end_date IS NULL
     OR now() BETWEEN start_date AND end_date)
```

Sample code 2. Driver retrieval based on distribution table

Leases can be stored in a table that has the same format as the distribution table. This table is used only for logging purposes, but also to retrieve client information when a lease must be renewed.

When a new driver needs to be added to the system, a new entry is inserted in the drivers table. Obsolete drivers can be disabled by either deleting them or setting the end_date to the current_date. Bootloaders that have a dedicated connection with the Server are notified immediately, others are upgraded as soon as their current lease has expired.

4.1.2 In-Database Drivolution Server

When implemented in the DBMS engine, the Drivolution Server directly responds to bootloader connections. It is possible to only allow connections through the Drivolution Server to ensure that client applications will only use drivers distributed by the DBMS. Code signing techniques can be used to ensure only certified drivers are used by the clients.

Alternatively, the Drivolution Server can listen on a different port than the database engine to allow legacy drivers to access the database using existing technology. Drivolution bootloader requests can then be served concurrently.

Most of the core functionality of the Drivolution Server code can be implemented in stored procedures to leverage existing database technologies.

4.1.3 External Drivolution Server

When the database does not support the Drivolution protocol or cannot be extended to support the Drivolution Server, it is possible to implement it as an external process querying the DBMS as a regular client application. Figure 2 shows how to implement a Drivolution Server with legacy databases.

In step 1, the Drivolution bootloader queries the Drivolution server. The server then connects to the database using a legacy database driver to return the appropriate driver to the bootloader

(step 3). Finally the bootloader can install the driver to connect to the database (step 4).

Even though this requires the Drivolution server to use a legacy driver, this solution has some benefits:

- When the legacy driver becomes obsolete, only the Drivolution server driver needs to be updated (that is a single machine) whereas no client machines require changes.
- When the legacy driver becomes obsolete, it means that the database has been upgraded, which is unlikely to happen without stopping the database. Therefore, the driver at the Drivolution server can be upgraded along with the database during the same planned downtime window.
- The Drivolution server can be upgraded without interrupting existing applications. If the Drivolution server is unavailable while a bootloader tries to renew its lease, the bootloader keeps its current implementation until the Drivolution server is restarted.

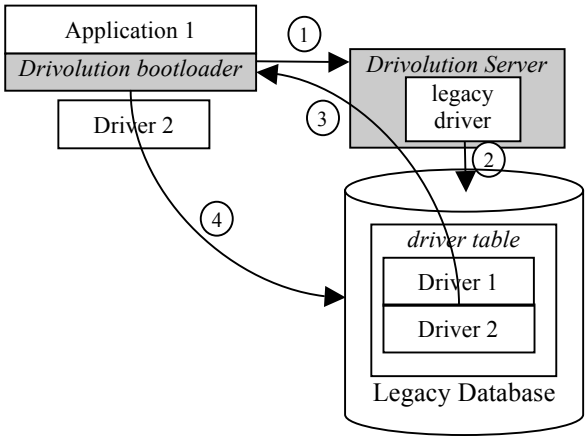

Figure 2. Drivolution server architecture for legacy databases

4.1.4 Standalone Drivolution Server

It is possible to have a single Drivolution server as a standalone service providing drivers for a set of databases. This scenario will be illustrated in 5.3.1. This can be useful in setups where databases do not support Drivolution natively or where an administrator wants to manage multiple database drivers from a centralized point.

An option to implement a standalone Drivolution server is to use an embedded database that does not require driver upgrades. As the rate of updates on the driver table is very low, it is easy to replicate the Drivolution server database for availability purposes.

5. CASE STUDIES

In this section, we present several use cases showing how Drivolution improves on the current state-of-the-art driver lifecycle. Section 5.1 shows how Drivolution can help DBAs administer heterogeneous DB systems; Section 5.2 presents a master/slave setup where Drivolution simplifies reconfiguration; Section 5.3 illustrates the use of Drivolution in replicated DB setups; and Section 5.4 describes two ways in which Drivolution can be used for customized driver delivery.

5.1 Simplifying Administration of Heterogeneous DBMSes

A database administrator (DBA) in large organizations is often responsible for a significant number of database instances. In such corporate environments, various applications use different database versions or even engines. If applications can have their own lifecycle, DBAs must share and use a common management infrastructure to administer all databases. This means that all possible drivers have to be installed and configured with the DBA management console.

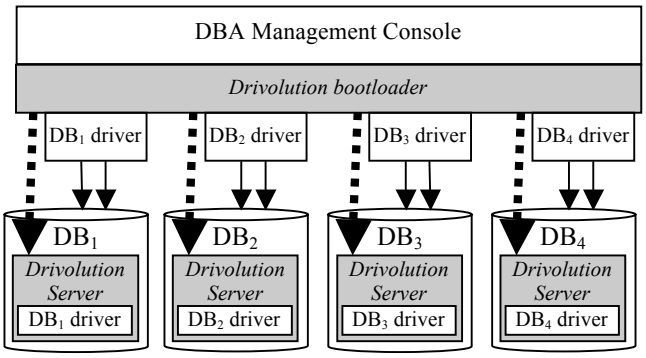

Figure 3. Configuration with complete native support for Drivolution

When all databases are fully Drivolution-compliant, a single Drivolution bootloader has to be installed in the management console. Figure 3 shows such a configuration. Each database automatically provides the appropriate driver for the platform that the management console is running on. The management console can access seamlessly any database without having to worry about driver configurations.

Table 5. Driver upgrades in a heterogeneous database for 2 DBAs with and without Drivolution

Tasks	Current State-of-the-Art	Drivolution
Accessing a new database	1. Download drivers for DBA_1 platform 2. Configure DBA_1 console to find driver 3. DBA_1 connects to db 4. Download drivers for DBA_2 platform 5. Configure DBA_2 console to find driver 6. DBA_2 connects to db	1. DBA_1 connects to db 2. DBA_2 connects to db
Database driver upgrade	1. Copy appropriate driver for DBA_1 platform 2. Remove DBA_1 old driver 3. Restart DBA_1 console 4. Copy right driver for DBA_2 platform 5. Remove DBA_2 old driver 6. Restart DBA_2 console	1. Insert drivers in database 2. Revoke old driver

Table 5 shows an example of the procedures that have to be performed by two DBAs for two administration tasks: access a new database from their console and upgrade the database driver. With Drivolution, the procedures are much shorter and simpler. The same procedure simplification would apply to any application connecting to the database.

In this configuration, driver upgrades are part of the database upgrade process. Each database can be upgraded independently of the others, without disturbing client applications. A tight integration of the Drivolution server with the database allows for additional compatibility checks to make sure that installed drivers are compatible with the database engine. This way, there is no possible confusion in installing drivers that are not supported by a given database.

As all applications fetch their driver from the database, it is impossible to forget to upgrade an application, as long as it uses Drivolution. If the driver upgrade contains important security upgrades, not only are the important applications upgraded, but so are the small management support scripts so often overlooked by DBAs that can also become security threats.

5.2 Dynamic Client Reconfiguration for Master/Slave Failover

Many organizations use master/slave configurations to achieve higher availability. When the master node needs to be stopped for maintenance operations, it is necessary to manually failover all client applications to the slave node. The failback operation must be applied to all database clients when the master is restarted. This process usually requires complex distributed application reconfiguration operations and is quite error-prone.

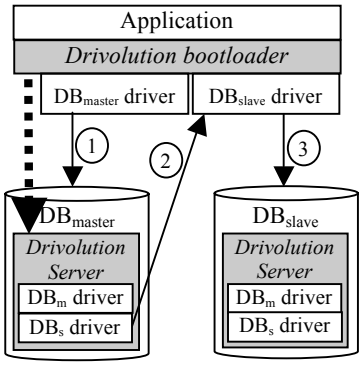

Figure 4. Dynamic client reconfiguration to operate a master/slave failover

Drivolution offers seamless driver upgrades to client applications. Instead of having one generic driver for all purposes, it is now possible to pre-generate a large number of pre-configured drivers to reconfigure client applications on-the-fly. Figure 4 shows a scenario where an application has to be reconfigured from a master to a slave database for a maintenance operation on the master node.

In this example, two drivers, DB_{master} and DB_{slave}, have been pre-generated to connect to the master and slave database, respectively. Whatever host name is found in the URL specified by the client application, it is ignored, and the drivers are pre-configured to always connect to the same database. The client URL is only used by the Drivolution bootloader to contact a Drivolution server.

As long as the master database is active, all applications are given the DB_{master} driver to connect to the master node (step 1 in Figure 4). When the master node needs to be stopped for maintenance, and the traffic must be redirected to the slave database, all applications have to be reconfigured. This can be easily performed by marking the DB_{master} driver as expired and providing the DB_{slave} driver as the new driver (step 2). All clients will upgrade their driver using this new driver, that will connect them to the slave database (step 3). Another driver upgrade from DB_{slave} to DB_{master} is used for the failback operation when the master becomes available again.

Drivolution offers a way to reconfigure simultaneously all applications from a single point. Drivers could be written in such a way that their configuration is generated on-the-fly by the database's Drivolution server and sent to the client. This way, client-side configuration is no longer needed. As client applications are usually in greater number than database instances, especially in replicated environments, this is an advantage.

5.3 Middleware-Based Database Replication

Sequoia [12] is an open source database replication middleware used in mission-critical production environments. Sequoia offers a JDBC driver with failover capabilities that needs to be installed in client applications. Sequoia drivers talk to replicated Sequoia controllers that implement the database clustering logic. Controllers use the database legacy JDBC drivers to access the database replicas. Sequoia can handle heterogeneous cluster configurations, regardless of whether the database engines have different versions but come from the same vendor, or are different engines from different vendors.

We have experimented with various configurations of Drivolution in Sequoia, corresponding to different real clustered application use cases. We show how driver deployments and upgrades are performed in these different scenarios.

5.3.1 Legacy Environment

When no component of the system supports Drivolution at all, it is necessary to use a dedicated Drivolution server that acts as a separate service to distribute drivers. Figure 5 gives an example of such configuration.

The client applications have to be configured to provide the bootloader with two connection URLs. One URL is used to contact the Drivolution server, and the other URL is passed to the actual driver implementation.

Sequoia driver upgrade: Sequoia uses its own wire protocol between drivers and controllers. Compatibility checking is done at connection time to ensure that protocol versions will work together. Drivers are backward compatible with older controllers. Sequoia drivers are also capable of automatic failover, so that they always end up connecting to a compatible controller, as long as one is available. By adding a new driver in the Drivolution server and making it available to all client applications, the cluster will upgrade automatically to this new version. If Sequoia controllers are stopped, upgraded and restarted one-by-one, drivers can be upgraded concurrently without any noticeable interruption for the application.

Database driver upgrade: If the cluster is homogeneous, it is possible to install a new driver for all replicas at once. Depending on the configuration, some databases (e.g., Sybase)

must use non-transactional persistent connections to be able to use features such as temporary tables. This implies that connections cannot be replaced before being closed. Therefore, nodes must be temporarily disabled and re-enabled to renew all connections around a consistent checkpoint. A good practice is to perform this operation on one node first, to check that the new driver is working properly. If the new driver does not work, it is possible to downgrade the driver by restoring the older version on the Drivolution server. Once the node has an operational driver, it can be re-enabled and resynchronized from its checkpoint by the Sequoia controller.

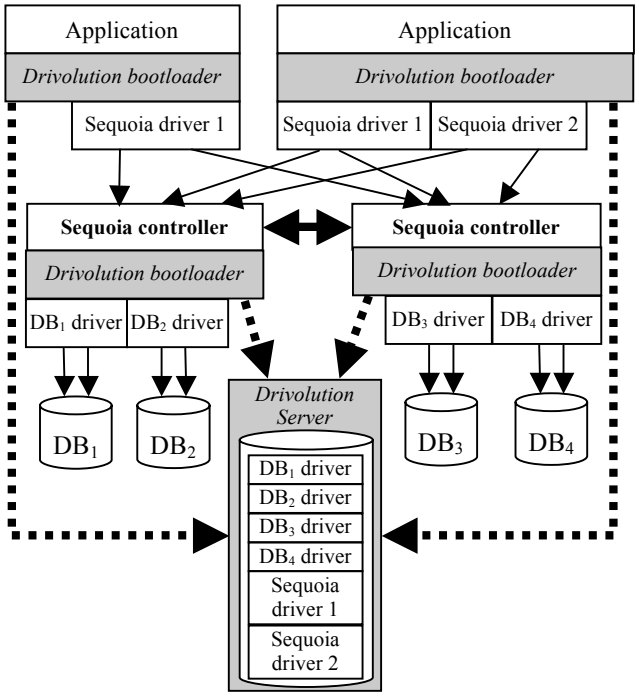

Figure 5. Standalone Drivolution server as a driver distribution service in the Sequoia the cluster

This configuration has the benefit of controlling drivers for all cluster resources from a single centralized point. However, this setup is sensitive to administrator errors, since it is easy to assign a wrong driver for a given resource. On the downside, the Drivolution server can become a single point of failure. It is then necessary to replicate it either using some hot-standby technique or active-active configurations even with weak consistency since updates to the Server are infrequent.

5.3.2 Highly Available Hybrid Setup

Since Sequoia controllers give applications the illusion that they are conversing with a single database, we also implemented a version of Drivolution for Sequoia controllers. Figure 6 shows a configuration where the Drivolution server is embedded in Sequoia controllers. The Server manages the drivers for both Sequoia clients and underlying database replicas. Unlike the previous configuration (5.3.1), the Drivolution server is replicated in each controller, preventing it from being a single point of failure. This implementation leverages the Sequoia replication infrastructure to synchronize Drivolution servers so as to always provide a consistent state.

Sequoia driver upgrade: In this Drivolution-compliant implementation, client applications do not need to use dual-URLs to specify the location of a remote Drivolution server. Sequoia JDBC URLs can contain multiple host names, as in 'jdbc:sequoia://controller1,controller2/db'. bootloaders exploit this information to load balance their requests and perform failover, if the first host in the list becomes unavailable. When a new driver is added to a Drivolution server, it is instantly replicated to other Drivolution servers. Therefore, all client applications can be upgraded no matter which server they are connected to.

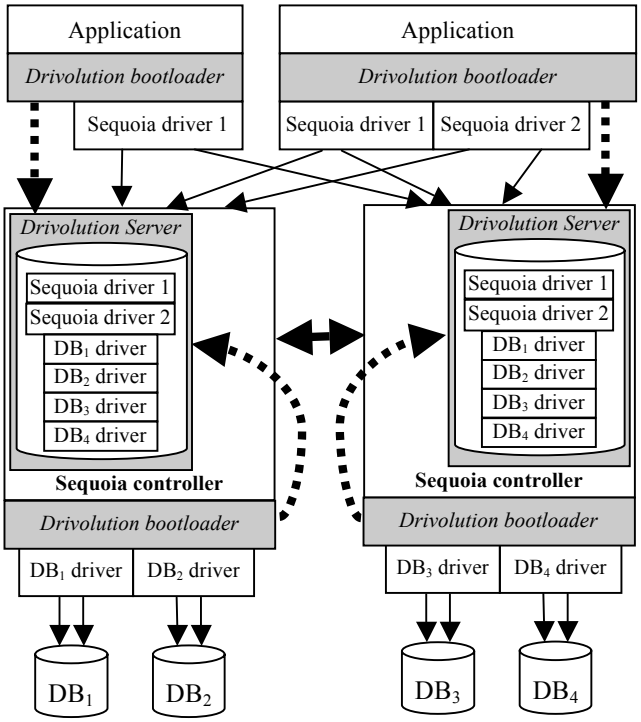

Figure 6. Drivolution servers embedded in Sequoia controllers

Database driver upgrade: Each bootloader installed in the controllers accesses the locally-embedded Drivolution server as if it were a standalone service for the database replicas. Each Server contains all drivers for all replicas in the cluster, which eases backend transfer between controllers for maintenance operations. Moreover, all database driver upgrades can be performed without interruption of the controller or changes in the configuration files, by simply making them available in the Drivolution server.

Sequoia is a Java middleware that already relies on JMX [14] for its management. Therefore, it is easy to integrate in a common console the management of Drivolution with the existing cluster management tools. Moreover, the embedded database approach used in our implementation allows easy integration with other applications.

5.4 Customized Driver Delivery

Some drivers are split into multiple packages that have to be configured separately, depending on which features the application requires. We describe how Drivolution can be used to hide this complex configuration from client applications.

5.4.1 Assembling Drivers on Demand

Most drivers externalize their localized messages in different internationalization packages. This is, for example the case for Oracle with a large NLS (National Language Support) package or Apache Derby with small packages per country. Drivolution servers can deliver the appropriate driver version with the exact required feature set to each application. These drivers can be stored statically in the database or be generated dynamically by aggregating packages. This prevents applications from loading an unnecessary large driver that contains features not used by the application.

If a PostgreSQL database contains a geographical database along with other regular databases, it is not necessary for all applications to get the GIS (Geographic Information System) extensions. Drivolution can help in providing only GIS clients with GIS extensions. The required extensions are statically encoded in the connection URL. However, the bootloader could also lazily detect a required extension through a ClassNotFoundException trapped by its classloader. The Drivolution server would then be contacted to provide the corresponding extension as an additional driver.

The DB2 JDBC driver installation guide [4] specifies that applications planning to use Kerberos security should add a set of 12 libraries: ibmjcefw.jar, ibmjlog.jar, etc. The Applications requiring Kerberos would get all these packages through Drivolution without any configuration. As drivers are loaded in a separate classloader, this also avoids any conflict with similar libraries required by other components of the applications.

5.4.2 Drivolution as a License Server

IBM DB2 has two licensing models: per-CPU and per-user. When using the per-user licensing model, each client application must use a license key that is provided in a separate jar file. Multiple strategies are possible to use Drivolution as a license management server.

Licenses can be statically assigned to clients so that each time a client connects it receives the same driver and license. This approach avoids any conflicts or starvation, but it is not very flexible. A more dynamic solution marks drivers as expired as soon as they have been delivered to a client. The bootloader can notify the Drivolution server when the driver is unloaded to give back its lease and to allow the driver to be re-used by another client. However, the Drivolution server must be able to detect when the client application terminates, to prevent drivers from holding a license forever.

If the Drivolution server and bootloader are using a dedicated connection, it can be used as a failure detector. If the Drivolution server is tightly integrated with the database, it can check if any connection with the client is still active in the database engine. Otherwise, the Drivolution server can wait for the client lease to expire and, if no lease renewal command has been issued by the bootloader, declare the driver freed.

In all scenarios, Drivolution can easily be extended as a central management location for database licenses required by client applications. Licenses can even be renewed or upgraded dynamically without having to interrupt client applications.

6. CONCLUSION

If DBMS vendors united behind a common standard wire protocol for the application-driver-DB engine communication, then driver lifecycle management would be easy. In the absence of such agreement, though, it is necessary to provide a standard bootstrap infrastructure that enables easy lifecycle management even in the presence of diverse legacy database drivers.

We described Drivolution, a new approach to distribute, install and upgrade database drivers, that is transparent to client applications. We implemented Drivolution for Java applications and JDBC-compliant database drivers. We have argued the benefits for the DBA's management tasks in heterogeneous environments or complex, highly available database clusters. Drivolution enables seamless dynamic reconfiguration of applications and delivery of custom drivers.

We believe it is feasible – even in the short term – for standard APIs, such as JDBC or ODBC, to provide a Drivolution bootloader that will be able to load any API-compliant driver. Language specific bootloaders can also be built, provided that dynamic code loading is available. With such Drivolution implementations, system administrators will be able to upgrade in one step hundreds of drivers in client applications from a single location with zero downtime.

7. ACKNOWLEDGEMENTS

We would like to thank Steve Dropsho, our shepherds TJ Giuli and Nick Briggs, and the anonymous reviewers for their valuable feedback. We are also thankful to the Continuent customers and staff who have inspired this work.

8. REFERENCES

[1] E. Cecchet and G. Candea – *Drivolution Developer's Guide version 1.0* – http://sourceforge.net/projects/drivolution/

[2] A. Descartes and Tim Bunce – *Programming the Perl DBI* – O'Reilly & Associates, 2000.

[3] Drivolution web site – http://sourceforge.net/projects/drivolution/

[4] IBM DB2 JDBC driver installation guide - http://publib.boulder.ibm.com/infocenter/db2luw/v8/index.jsp?topic=/com.ibm.db2.udb.doc/ad/t0010264.htm.

[5] S. Liang and Gilad Bracha - *Dynamic Class Loading in the Java Virtual Machine* – OOPSLA'98, October 1998.

[6] Microsoft White Paper – *Match.com Halves Server Farm, Doubles Speed with ASP.NET and Windows Server* – http://download.microsoft.com/documents/customerevidence/ 6740_Match.com.doc, 2004.

[7] MySQL Enterprise drivers – http://www.mysql.com/products/connector/

[8] MySQL supported platforms – http://www.mysql.com/support/supportedplatforms/

[9] S. Newman and Jim Gray – *Which Way to Remote SQL?* – Database Programming and Design, v4.2, Dec. 1991.

[10] James Norton – *Dynamic Class Loading for C++ on Linux* – Linux Journal, May 2000.

[11] Pair networks - http://www.pair.com/support/knowledge_base/our_network_and_servers/ server_configurations.html

[12] Sequoia Project. http://sequoiadb.sourceforge.net/

[13] R. Strahl – *Dynamically executing code in .Net* – West Wind whitepaper, http://www.west-wind.com/, 2002.

[14] Sun Microsystems – *Java™ Management Extensions Instrumentation and Agent Specification, v1.2* – 2002